CREATE INDEX

Purpose

Use the `CREATE INDEX` statement to create an index on:

- One or more columns of a table, a partitioned table, an index-organized table, or a cluster
- One or more scalar typed object attributes of a table or a cluster
- A nested table storage table for indexing a nested table column

An **index** is a schema object that contains an entry for each value that appears in the indexed column(s) of the table or cluster and provides direct, fast access to rows. Oracle Database supports several types of index:

- Normal indexes. (By default, Oracle Database creates B-tree indexes.)
- **Bitmap indexes**, which store rowids associated with a key value as a bitmap
- **Partitioned indexes**, which consist of partitions containing an entry for each value that appears in the indexed column(s) of the table
- **Function-based indexes**, which are based on expressions. They enable you to construct queries that evaluate the value returned by an expression, which in turn may include built-in or user-defined functions.
- **Domain indexes**, which are instances of an application-specific index of type *`indextype`*

See Also:

- *Oracle Database Concepts*for a discussion of indexes
- ALTER INDEXand DROP INDEX

Additional Topics

Prerequisites

To create an index in your own schema, one of the following conditions must be true:

- The table or cluster to be indexed must be in your own schema.
- You must have the `INDEX` object privilege on the table to be indexed.
- You must have the `CREATE ANY INDEX` system privilege.

To create an index in another schema, you must have the `CREATE ANY INDEX` system privilege. Also, the owner of the schema to contain the index must have either the `UNLIMITED TABLESPACE` system privilege or space quota on the tablespaces to contain the index or index partitions.

To create a domain index in your own schema, in addition to the prerequisites for creating a conventional index, you must also have the `EXECUTE` object privilege on the indextype. If you are creating a domain index in another user's schema, then the index owner also must have the `EXECUTE` object privilege on the indextype and its underlying implementation type. Before creating a domain index, you should first define the indextype.

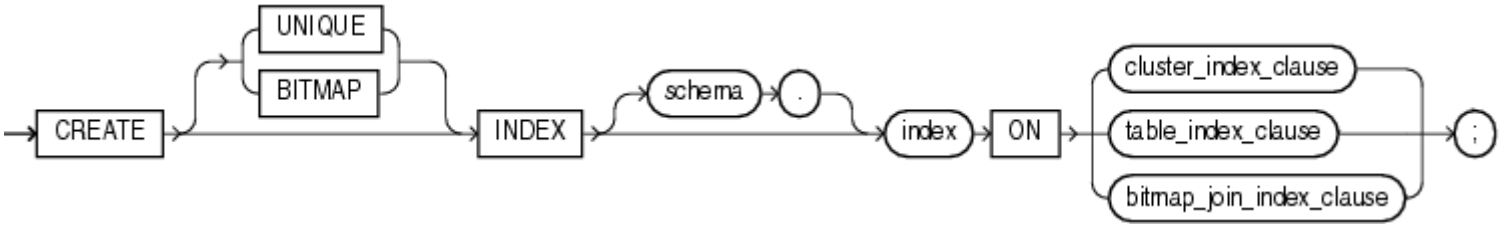
To create a function-based index, in addition to the prerequisites for creating a conventional index, if the index is based on user-defined functions, then those functions must be marked `DETERMINISTIC`. Also, you must have the `EXECUTE` object privilege on any user-defined function(s) used in the function-based index if those functions are owned by another user.

See Also:

CREATE INDEXTYPE

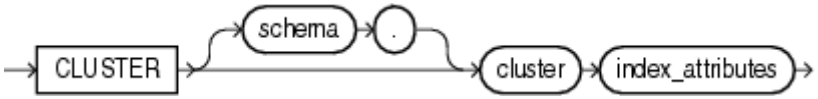
Syntax

***create_index*::=**



Description of the illustration create_index.gif

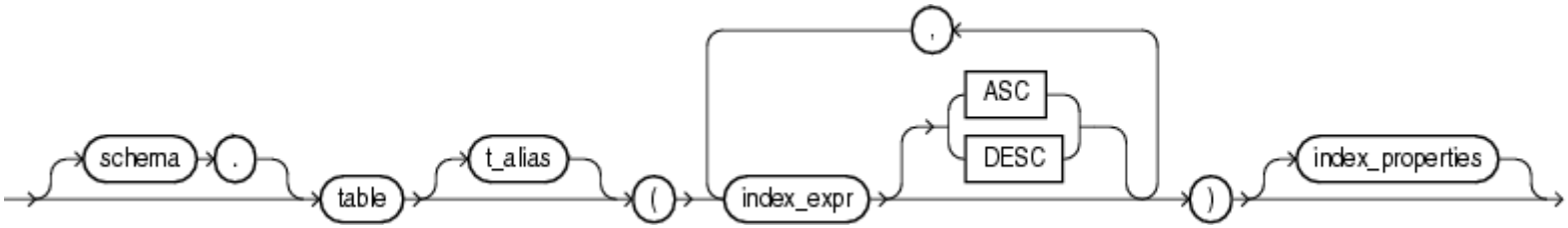
***cluster_index_clause* ::=**



Description of the illustration cluster_index_clause.gif

(*index_attributes::=*)

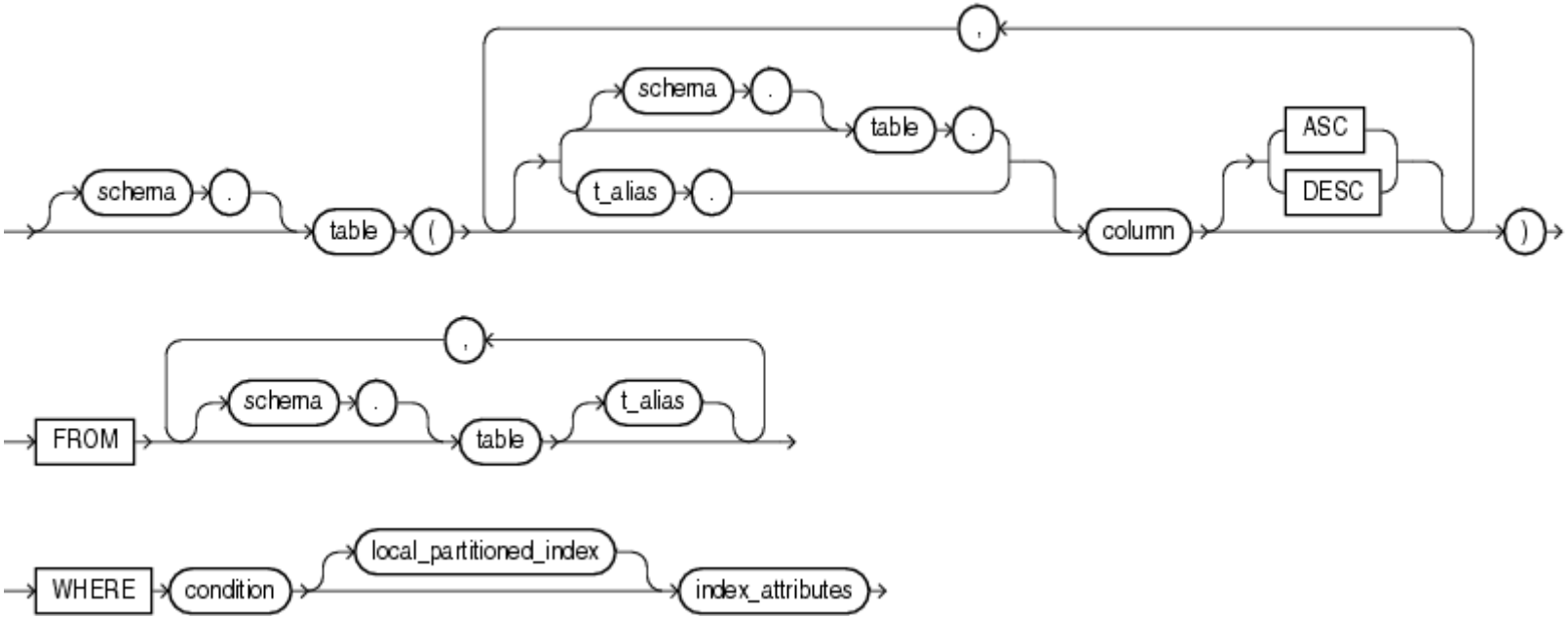
***table_index_clause*::=**



Description of the illustration table_index_clause.gif

(*index_properties::=*)

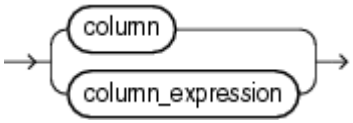
***bitmap_join_index_clause*::=**



Description of the illustration bitmap_join_index_clause.gif

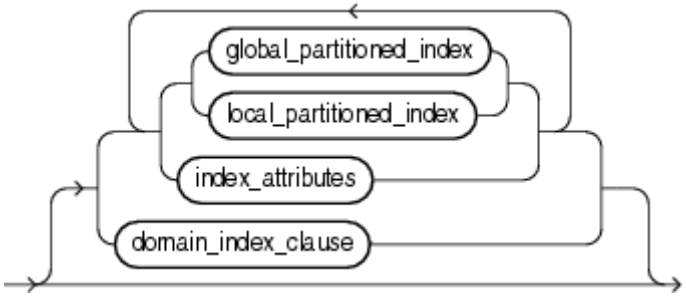
(*local_partitioned_index* ::=, *index_attributes*::=)

index_expr::=



Description of the illustration index_expr.gif

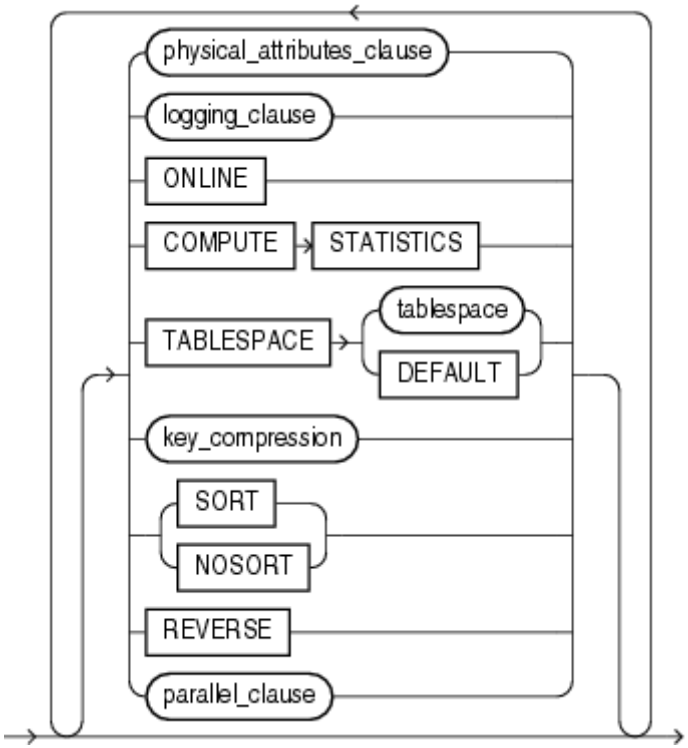
index_properties::=



Description of the illustration index_properties.gif

(*global_partitioned_index ::=*, *local_partitioned_index ::=*, *index_attributes::=*, *domain_index_clause ::=*)

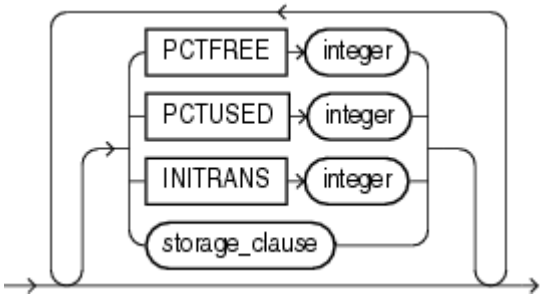
index_attributes::=



Description of the illustration index_attributes.gif

(*physical_attributes_clause ::=*, *logging_clause::=*, *key_compression=*, *parallel_clause::=*)

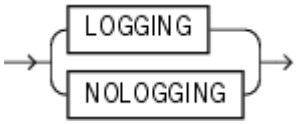
physical_attributes_clause::=



Description of the illustration physical_attributes_clause.gif

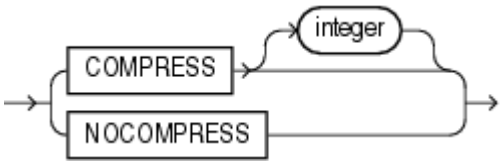
(*storage_clause*::=)

logging_clause::=



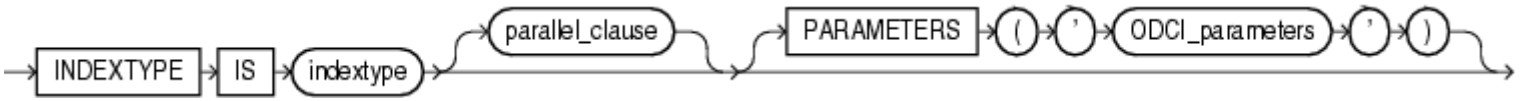
Description of the illustration logging_clause.gif

key_compression=



Description of the illustration key_compression.gif

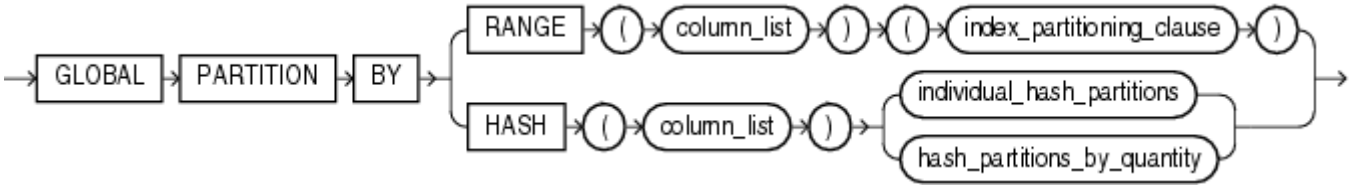
domain_index_clause::=



Description of the illustration domain_index_clause.gif

(*parallel_clause::=*)

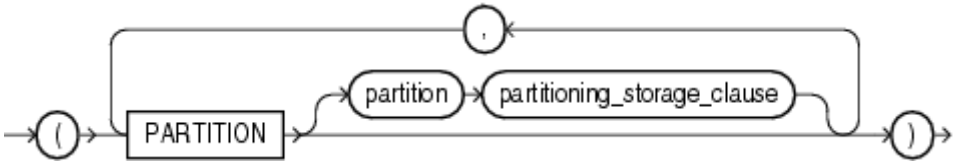
global_partitioned_index ::=



Description of the illustration global_partitioned_index.gif

(*index_partitioning_clause::=*, *individual_hash_partitions::=*, *hash_partitions_by_quantity::=*)

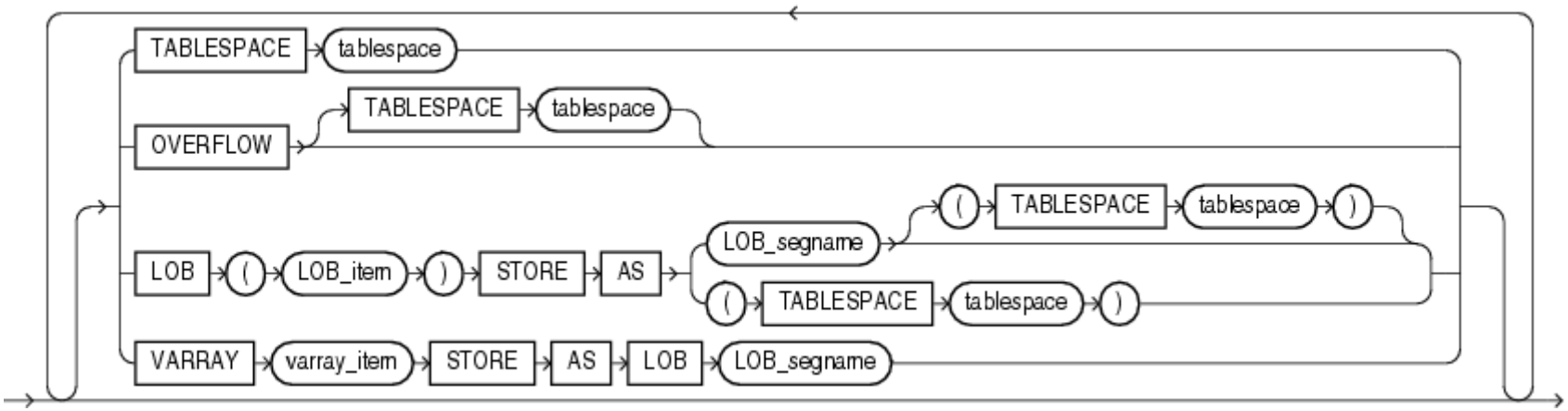
individual_hash_partitions::=



Description of the illustration individual_hash_partitions.gif

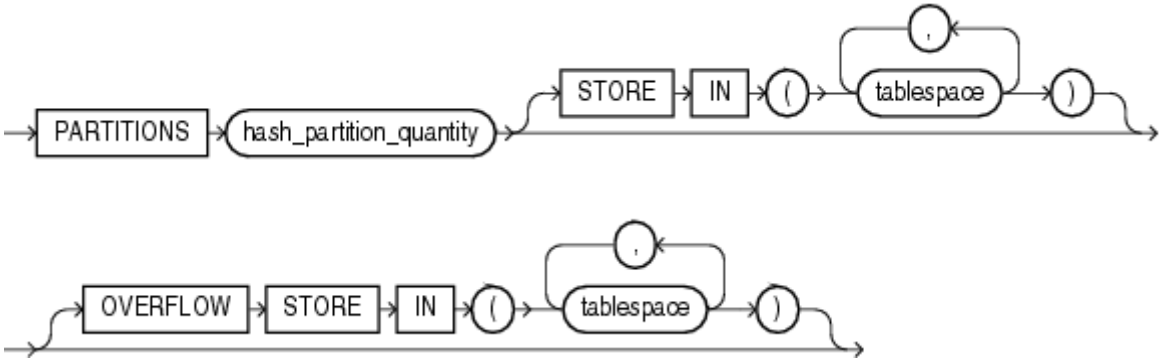
(*partitioning_storage_clause::=*)

partitioning_storage_clause::=



Description of the illustration partitioning_storage_clause.gif

***hash_partitions_by_quantity*::=**



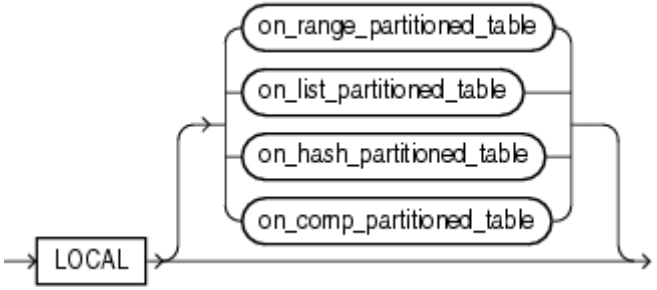
Description of the illustration hash_partitions_by_quantity.gif

***index_partitioning_clause*::=**

Description of the illustration index_partitioning_clause.gif

(*segment_attributes_clause*::=)

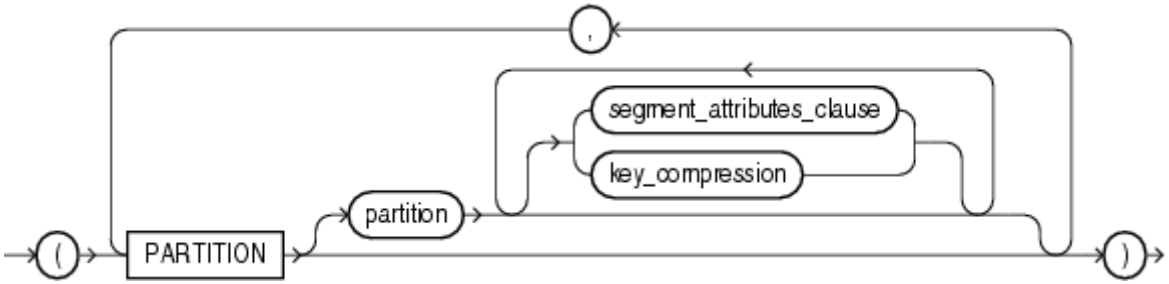
local_partitioned_index::=



Description of the illustration local_partitioned_index.gif

(*on_range_partitioned_table ::=*, *on_list_partitioned_table::=*, *on_hash_partitioned_table ::=*, *on_comp_partitioned_table::=*)

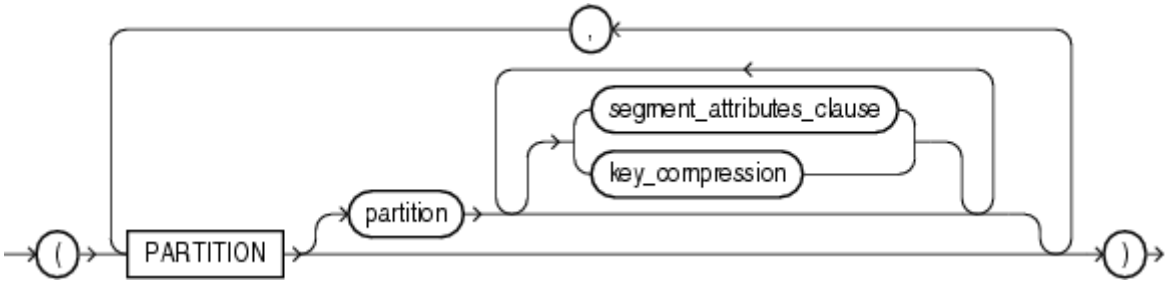
on_range_partitioned_table::=



Description of the illustration on_range_partitioned_table.gif

(*segment_attributes_clause::=*)

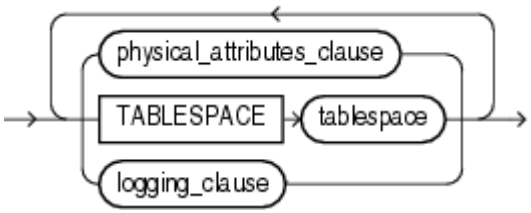
on_list_partitioned_table::=



Description of the illustration on_list_partitioned_table.gif

(*segment_attributes_clause::=*)

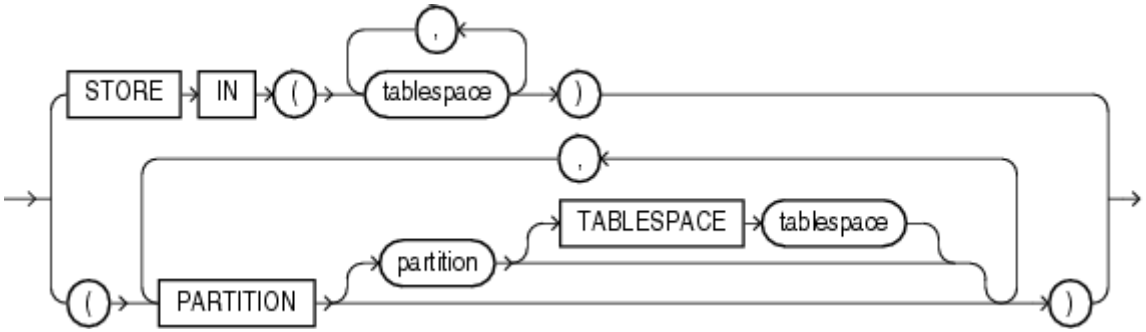
segment_attributes_clause::=



Description of the illustration segment_attributes_clause.gif

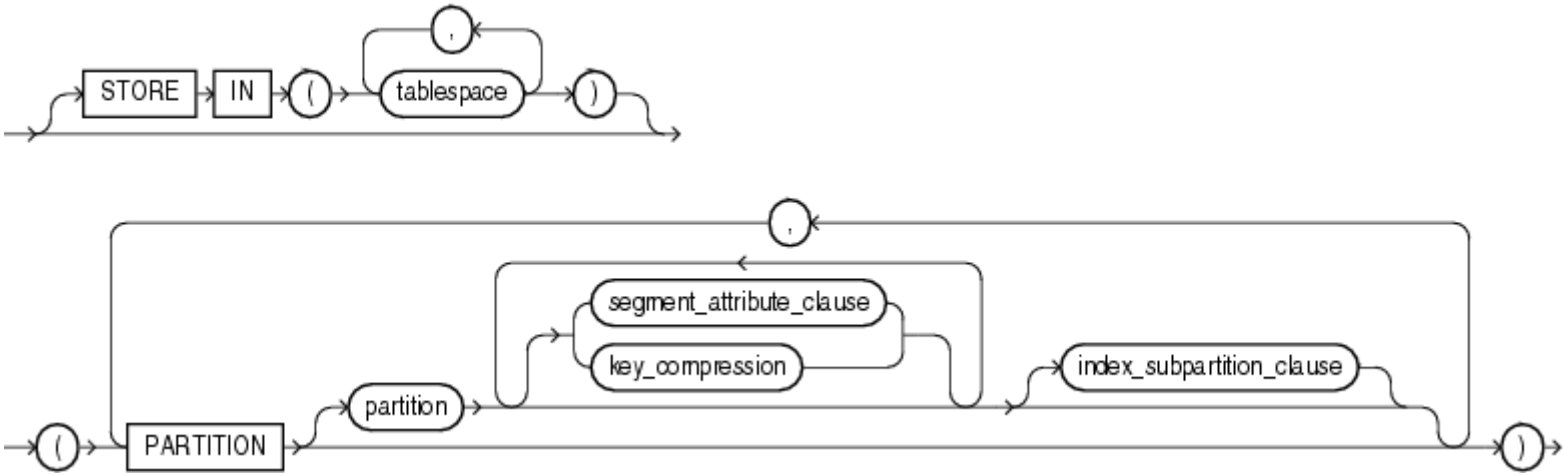
(*physical_attributes_clause ::=*, *logging_clause::=*

on_hash_partitioned_table::=



Description of the illustration on_hash_partitioned_table.gif

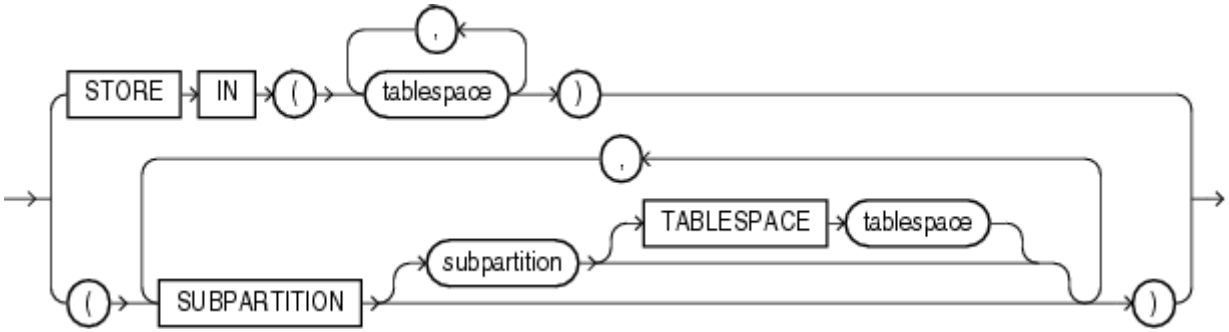
***on_comp_partitioned_table*::=**



Description of the illustration on_comp_partitioned_table.gif

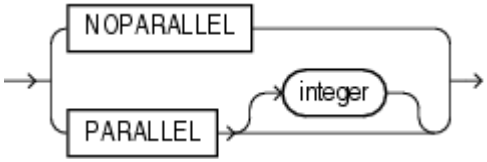
(*segment_attributes_clause*::=, *index_subpartition_clause*::=)

***index_subpartition_clause*::=**



Description of the illustration index_subpartition_clause.gif

***parallel_clause*::=**



Description of the illustration parallel_clause.gif

(*storage_clause::=*)

Semantics

UNIQUE

Specify `UNIQUE` to indicate that the value of the column (or columns) upon which the index is based must be unique.

Restrictions on Unique Indexes
Unique indexes are subject to the following restrictions:

- You cannot specify both `UNIQUE` and `BITMAP`.
- You cannot specify `UNIQUE` for a domain index.

> **See Also:**
>
> "Unique Constraints"for information on the conditions that satisfy a unique constraint

BITMAP

Specify `BITMAP` to indicate that *`index`* is to be created with a bitmap for each distinct key, rather than indexing each row separately. Bitmap indexes store the rowids associated with a key value as a bitmap. Each bit in the bitmap corresponds to a possible rowid. If the bit is set, then it means that the row with the corresponding rowid contains the key value. The internal representation of bitmaps is best suited for applications with low levels of concurrent transactions, such as data warehousing.

> **Note:**

> Oracle does not index table rows in which all key columns are null except in the case of bitmap indexes. Therefore, if you want an index on all rows of a table, then you must either specify `NOT NULL` constraints for the index key columns or create a bitmap index.

Restrictions on Bitmap Indexes
Bitmap indexes are subject to the following restrictions:

- You cannot specify `BITMAP` when creating a global partitioned index.
- You cannot create a bitmap secondary index on an index-organized table unless the index-organized table has a mapping table associated with it.
- You cannot specify both `UNIQUE` and `BITMAP`.
- You cannot specify `BITMAP` for a domain index.

See Also:

- *Oracle Database Concepts* and *Oracle Database Performance Tuning Guide* for more information about using bitmap indexes
- CREATE TABLE for information on mapping tables
- "Bitmap Index Example"

schema

Specify the schema to contain the index. If you omit *`schema`*, then Oracle Database creates the index in your own schema.

index

Specify the name of the index to be created.

See Also:

"Creating an Index: Example" and "Creating an Index on an XMLType Table: Example"

cluster_index_clause

Use the *cluster_index_clause* to identify the cluster for which a cluster index is to be created. If you do not qualify cluster with *schema*, then Oracle Database assumes the cluster is in your current schema. You cannot create a cluster index for a hash cluster.

> **See Also:**
>
> CREATE CLUSTERand "Creating a Cluster Index: Example"

table_index_clause

Specify the table on which you are defining the index. If you do not qualify *table* with *schema*, then Oracle Database assumes the table is contained in your own schema.

You create an index on a nested table column by creating the index on the nested table storage table. Include the NESTED_TABLE_ID pseudocolumn of the storage table to create a UNIQUE index, which effectively ensures that the rows of a nested table value are distinct.

> **See Also:**
>
> "Indexes on Nested Tables: Example"

You can perform DDL operations (such as ALTER TABLE, DROP TABLE, CREATE INDEX) on a temporary table only when no session is bound to it. A session becomes bound to a temporary table by performing an INSERT operation on the table. A session becomes unbound to the temporary table by issuing a TRUNCATE statement or at session termination, or, for a transaction-specific temporary table, by issuing a COMMIT or ROLLBACK statement.

Restrictions on the *table_index_clause*
This clause is subject to the following restrictions:

- If *index* is locally partitioned, then *table* must be partitioned.

- If *table* is index-organized, this statement creates a secondary index. The index contains the index key and the logical rowid of the index-organized table. The logical rowid excludes columns that are also part of the index key. You cannot specify REVERSE for this secondary index, and the combined size of the index key and the logical rowid should be less than the block size.

- If *table* is a temporary table, then *index* will also be temporary with the same scope (session or transaction) as *table*. The following restrictions apply to indexes on temporary tables:
  - The only part of *index_properties* you can specify is *index_attributes*.
  - Within *index_attributes*, you cannot specify the *physical_attributes_clause*, the *parallel_clause*, the *logging_clause*, or TABLESPACE.
  - You cannot create a domain index on a temporary table.

> **See Also:**
>
> CREATE TABLEand *Oracle Database Concepts*for more information on temporary tables

t_alias

Specify a correlation name (alias) for the table upon which you are building the index.

> **Note:**
>
> This alias is required if the *index_expr* references any object type attributes or object type methods. See "Creating a Function-based Index on a Type Method: Example"and "Indexing on Substitutable Columns: Examples".

index_expr

For *index_expr*, specify the column or column expression upon which the index is based.

column
Specify the name of one or more columns in the table. A bitmap index can have a maximum of 30 columns. Other indexes can have as many as 32 columns. These columns define the **index key**.

If the index is local nonprefixed (see *local_partitioned_index*), then the index key must contain the partitioning key.

You can create an index on a scalar object attribute column or on the system-defined `NESTED_TABLE_ID` column of the nested table storage table. If you specify an object attribute column, then the column name must be qualified with the table name. If you specify a nested table column attribute, then it must be qualified with the outermost table name, the containing column name, and all intermediate attribute names leading to the nested table column attribute.

Restriction on Index Columns
You cannot create an index on columns or attributes whose type is user-defined, `LONG`, `LONG RAW`, LOB, or `REF`, except that Oracle Database supports an index on `REF` type columns or attributes that have been defined with a `SCOPE` clause.

column_expression
Specify an expression built from columns of *`table`*, constants, SQL functions, and user-defined functions. When you specify *`column_expression`*, you create a **function-based index**.

See Also:

"Notes on Function-based Indexes", "Restrictions on Function-based Indexes", and "Function-Based Index Examples"

Name resolution of the function is based on the schema of the index creator. User-defined functions used in *`column_expression`* are fully name resolved during the `CREATE INDEX` operation.

After creating a function-based index, collect statistics on both the index and its base table using the `DBMS_STATS` package. Such statistics will enable Oracle Database to correctly decide when to use the index.

Function-based unique indexes can be useful in defining a conditional unique constraint on a column or combination of columns. Please refer to "Using a Function-based Index to Define Conditional Uniqueness: Example"for an example.

See Also:

*PL/SQL Packages and Types Reference*for more information on the `DBMS_STATS` package

Notes on Function-based Indexes
The following notes apply to function-based indexes:

- When you subsequently query a table that uses a function-based index, you must ensure in the query that is not null. However, Oracle Database will use a function-based index in a query even if the columns specified in the `WHERE` clause are in a different order than their order in the *`column_expression`*that defined the function-based index.

> **See Also:**
>
> "Function-Based Index Examples"

- If the function on which the index is based becomes invalid or is dropped, then Oracle Database marks the index `DISABLED`. Queries on a `DISABLED`index fail if the optimizer chooses to use the index. DML operations on a `DISABLED` index fail unless the index is also marked `UNUSABLE` **and** the parameter `SKIP_UNUSABLE_INDEXES` is set to `true`. Please refer to ALTER SESSIONfor more information on this parameter.

- If a public synonym for a function, package, or type is used in *`column_expression`*, and later an actual object with the same name is created in the table owner's schema, then Oracle Database disables the function-based index. When you subsequently enable the function-based index using `ALTER INDEX... ENABLE` or `ALTER INDEX ... REBUILD`, the function, package, or type used in the *`column_expression`* continues to resolve to the function, package, or type to which the public synonym originally pointed. It will not resolve to the new function, package, or type.

- If the definition of a function-based index generates internal conversion to character data, then use caution when changing NLS parameter settings. Function-based indexes use the current database settings for NLS parameters. If you reset these parameters at the session level, then queries using the function-based index may return incorrect results. Two exceptions are the collation parameters (`NLS_SORT` and `NLS_COMP`). Oracle Database handles the conversions correctly even if these have been reset at the session level.

Restrictions on Function-based Indexes
Function-based indexes are subject to the following restrictions:

- The value returned by the function referenced in *`column_expression`* is subject to the same restrictions as are the index columns of a B-tree index. Please refer to "Restriction on Index Columns".

- Any user-defined function referenced in *`column_expression`* must be declared as `DETERMINISTIC`.

- For a function-based globally partitioned index, the *`column_expression`* cannot be the partitioning key.

- The *`column_expression`* can be any form of expression except a scalar subquery expression.

- All functions must be specified with parentheses, even if they have no parameters. Otherwise Oracle Database interprets them as column names.

- Any function you specify in *`column_expression`* must return a repeatable value. For example, you cannot specify the `SYSDATE` or `USER` function or the `ROWNUM` pseudocolumn.

- The *column_expression* cannot contain any aggregate functions.

See Also:

CREATE FUNCTIONand *PL/SQL User's Guide and Reference*

ASC | DESC

Use ASC or DESC to indicate whether the index should be created in ascending or descending order. Indexes on character data are created in ascending or descending order of the character values in the database character set.

Oracle Database treats descending indexes as if they were function-based indexes. As with other function-based indexes, the database does not use descending indexes until you first analyze the index and the table on which the index is defined. See the *column_expression*clause of this statement.

Ascending unique indexes allow multiple NULL values. However, in descending unique indexes, multiple NULL values are treated as duplicate values and therefore are not permitted.

Restriction on Ascending and Descending Indexes

You cannot specify either of these clauses for a domain index. You cannot specify DESC for a reverse index. Oracle Database ignores DESC if *index* is bitmapped or if the COMPATIBLE initialization parameter is set to a value less than 8.1.0.

index_attributes

Specify the optional index attributes.

physical_attributes_clause

Use the *physical_attributes_clause* to establish values for physical and storage characteristics for the index.

If you omit this clause, then Oracle Database sets PCTFREE to 10 and INITRANS to 2.

Restriction on Index Physical Attributes

You cannot specify the PCTUSED parameter for an index.

See Also:

*physical_attributes_clause*and *storage_clause*for a complete description of these clauses

TABLESPACE
For *tablespace*, specify the name of the tablespace to hold the index, index partition, or index subpartition. If you omit this clause, then Oracle Database creates the index in the default tablespace of the owner of the schema containing the index.

For a local index, you can specify the keyword DEFAULT in place of *tablespace*. New partitions or subpartitions added to the local index will be created in the same tablespace(s) as the corresponding partitions or subpartitions of the underlying table.

key_compression
Specify COMPRESS to enable key compression, which eliminates repeated occurrence of key column values and may substantially reduce storage. Use *integer* to specify the prefix length (number of prefix columns to compress).

- For unique indexes, the valid range of prefix length values is from 1 to the number of key columns minus 1. The default prefix length is the number of key columns minus 1.

- For nonunique indexes, the valid range of prefix length values is from 1 to the number of key columns. The default prefix length is the number of key columns.

  Oracle Database compresses only nonpartitioned indexes that are nonunique or unique indexes of at least two columns.

Restriction on Key Compression
You cannot specify COMPRESS for a bitmap index.

See Also:

"Compressing an Index: Example"

NOCOMPRESS
Specify NOCOMPRESS to disable key compression. This is the default.

SORT | NOSORT

By default, Oracle Database sorts indexes in ascending order when it creates the index. You can specify NOSORT to indicate to the database that the rows are already stored in the database in ascending order, so that Oracle Database does not have to sort the rows when creating the index. If the rows of the indexed column or columns are not stored in ascending order, then the database returns an error. For greatest savings of sort time and space, use this clause immediately after the initial load of rows into a table. If you specify neither of these keywords, then SORT is the default.

Restrictions on NOSORT
This parameter is subject to the following restrictions:

- You cannot specify REVERSE with this clause.
- You cannot use this clause to create a cluster index partitioned or bitmap index.
- You cannot specify this clause for a secondary index on an index-organized table.

REVERSE
Specify REVERSE to store the bytes of the index block in reverse order, excluding the rowid.

Restrictions on Reverse Indexes
Reverse indexes are subject to the following restrictions:

- You cannot specify NOSORT with this clause.
- You cannot reverse a bitmap index or an index on an index-organized table.

logging_clause
Specify whether the creation of the index will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file. This setting also determines whether subsequent Direct Loader (SQL*Loader) and direct-path INSERT operations against the index are logged or not logged. LOGGING is the default.

If *index* is nonpartitioned, then this clause specifies the logging attribute of the index.

If *index* is partitioned, then this clause determines:

- The default value of all partitions specified in the CREATE statement, unless you specify the *logging_clause* in the PARTITION description clause

- The default value for the segments associated with the index partitions
- The default value for local index partitions or subpartitions added implicitly during subsequent `ALTER TABLE ... ADD PARTITION` operations

The logging attribute of the index is independent of that of its base table.

If you omit this clause, then the logging attribute is that of the tablespace in which it resides.

See Also:

- *logging_clause*for a full description of this clause
- *Oracle Database Concepts*and *Oracle Data Warehousing Guide*for more information about logging and parallel DML
- "Creating an Index in NOLOGGING Mode: Example"

ONLINE
Specify `ONLINE` to indicate that DML operations on the table will be allowed during creation of the index.

Restrictions on Online Index Building
Online index building is subject to the following restrictions:

- Parallel DML is not supported during online index building. If you specify `ONLINE` and then issue parallel DML statements, then Oracle Database returns an error.
- You cannot specify `ONLINE` for a bitmap index or a cluster index.
- You cannot specify `ONLINE` for a conventional index on a `UROWID` column.
- For a nonunique secondary index on an index-organized table, the number of index key columns plus the number of primary key columns that are included in the logical rowid in the index-organized table cannot exceed 32. The logical rowid excludes columns that are part of the index key.

See Also:

Oracle Database Concepts for a description of online index building and rebuilding

COMPUTE STATISTICS
In earlier releases, you could use this clause to start or stop the collection of statistics on an index. This clause has been deprecated. Oracle Database now automatically collects statistics during index creation and rebuild. This clause is supported for backward compatibility and will not cause errors.

Restriction on COMPUTE STATISTICS Clause
You cannot specify this clause for a domain index.

parallel_clause

Specify the *parallel_clause* if you want creation of the index to be parallelized.

For complete information on this clause, please refer to *parallel_clause* in the documentation on `CREATE TABLE`.

Index Partitioning Clauses

Use the *global_partitioned_index* clause and the *local_partitioned_index* clauses to partition *index*.

The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Please refer to *Oracle Database Administrator's Guide* for a discussion of these restrictions.

See Also:

"Partitioned Index Examples"

global_partitioned_index

The *global_partitioned_index* clause lets you specify that the partitioning of the index is user defined and is not equipartitioned with the underlying table. By default, nonpartitioned indexes are global indexes.

You can partition a global index by range or by hash. In both cases, you can specify up to 32 columns as partitioning key columns. The partitioning column list must specify a left prefix of the index column list. That is, if the index is defined on columns `a`, `b`, and `c`, then for the columns you can specify `(a, b, c)`, or `(a, b)`, or `(a, c)`, but you cannot specify `(b, c)` or `(c)` or `(b, a)`. If you omit the partition names, then Oracle Database assigns names of the form `SYS_Pn`.

GLOBAL PARTITION BY RANGE
Use this clause to create a range-partitioned global index. Oracle Database will partition the global index on the ranges of values from the table columns you specify in the column list.

See Also:

"Creating a Range-Partitioned Global Index: Example"

GLOBAL PARTITION BY HASH
Use this clause to create a hash-partitioned global index. Oracle Database assigns rows to the partitions using a hash function on values in the partitioning key columns.

See Also:

the `CREATE TABLE` clause *hash_partitioning*for information on the two methods of hash partitioning and "Creating a Hash-Partitioned Global Index: Example"

Restrictions on Global Partitioned Indexes
Global partitioned indexes are subject to the following restrictions:

- The partitioning key column list cannot contain the `ROWID` pseudocolumn or a column of type `ROWID`.
- The only property you can specify for hash partitions is tablespace storage. Therefore, you cannot specify LOB or varray storage clauses in the *`partitioning_storage_clause`* of *`individual_hash_partitions`*.
- You cannot specify the `OVERFLOW` clause of *`hash_partitions_by_quantity`*, as that clause is valid only for index-organized table partitions.

Note:

If your enterprise has or will have databases using different character sets, then use caution when partitioning on character columns. The sort sequence of characters is not identical in all character sets.

See Also:

*Oracle Database Globalization Support Guide*for more information on character set support

index_partitioning_clause

Use this clause to describe the individual index partitions. The number of repetitions of this clause determines the number of partitions. If you omit *partition*, then Oracle Database generates a name with the form SYS_Pn.

For VALUES LESS THAN (*value_list*), specify the noninclusive upper bound for the current partition in a global index. The value list is a comma-delimited, ordered list of literal values corresponding to the column list in the *global_partitioned_index* clause. Always specify MAXVALUE as the value of the last partition.

> **Note:**
> If the index is partitioned on a DATE column, and if the date format does not specify the first two digits of the year, then you must use the TO_DATEfunction with a 4-character format mask for the year. The date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT. Please refer to *Oracle Database Globalization Support Guide*for more information on these initialization parameters.

See Also:

"Range Partitioning Example"

local_partitioned_index

The *local_partitioned_index* clauses let you specify that the index is partitioned on the same columns, with the same number of partitions and the same partition bounds as *table*. Oracle Database automatically maintains local index partitioning as the underlying table is repartitioned.

on_range_partitioned_table
This clause lets you specify the names and attributes of index partitions on a range-partitioned table. If you specify this clause, then the number of PARTITION clauses must be equal to the number of table partitions, and in the same order. If you omit *partition*, then Oracle Database generates a name that is consistent with the corresponding table partition. If the name conflicts with an existing index partition name, then the database uses the form SYS_Pn.

You cannot specify key compression for an index partition unless you have specified key compression for the index.

on_list_partitioned_table
The *on_list_partitioned_table* clause is identical to *on_range_partitioned_table*.

on_hash_partitioned_table
This clause lets you specify names and tablespace storage for index partitions on a hash-partitioned table.

If you specify any PARTITION clauses, then the number of these clauses must be equal to the number of table partitions. If you omit *partition*, then Oracle Database generates a name that is consistent with the corresponding table partition. If the name conflicts with an existing index partition name, then the database uses the form SYS_P*n*. You can optionally specify tablespace storage for one or more individual partitions. If you do not specify tablespace storage either here or in the STORE IN clause, then the database stores each index partition in the same tablespace as the corresponding table partition.

The STORE IN clause lets you specify one or more tablespaces across which Oracle Database will distribute all the index hash partitions. The number of tablespaces need not equal the number of index partitions. If the number of index partitions is greater than the number of tablespaces, then the database cycles through the names of the tablespaces.

on_comp_partitioned_table
This clause lets you specify the name and tablespace storage of index partitions on a composite-partitioned table.

The STORE IN clause is valid only for range-hash composite-partitioned tables. It lets you specify one or more default tablespaces across which Oracle Database will distribute all index hash subpartitions. You can override this storage by specifying different tablespace storage for the subpartitions of an individual partition in the second STORE IN clause in the *index_subpartition_clause*.

For range-list composite-partitioned tables, you can specify default tablespace storage for the list subpartitions in the PARTITION clause. You can override this storage by specifying different tablespace storage for the list subpartitions of an individual partition in the SUBPARTITION clause of the *index_subpartition_clause*.

You cannot specify key compression for an index partition unless you have specified key compression for the index.

index_subpartition_clause
This clause lets you specify names and tablespace storage for index subpartitions in a composite-partitioned table.

The STORE IN clause is valid only for hash subpartitions of a range-hash composite-partitioned table. It lets you specify one or more tablespaces across which Oracle Database will distribute all the index hash subpartitions. The SUBPARTITION clause is valid for subpartitions of both range-hash and range-list composite-partitioned tables.

If you specify any SUBPARTITION clauses, then the number of those clauses must be equal to the number of table subpartitions. If you omit *subpartition*, then the database generates a name that is consistent with the corresponding table subpartition. If the name conflicts with an existing index subpartition name, then the database uses the form SYS_SUBP*n*.

The number of tablespaces need not equal the number of index subpartitions. If the number of index subpartitions is greater than the number of tablespaces, then the database cycles through the names of the tablespaces.

If you do not specify tablespace storage for subpartitions either in the *on_comp_partitioned_table* clause or in the *index_subpartition_clause*, then Oracle Database uses the tablespace specified for *index*. If you also do not specify tablespace storage for *index*, then the database stores the subpartition in the same tablespace as the corresponding table subpartition.

domain_index_clause

Use the *domain_index_clause* to indicate that *index* is a domain index, which is an instance of an application-specific index of type *indextype*.

Creating a domain index requires a number of preceding operations. You must first create an implementation type for an indextype. You must also create a functional implementation and then create an operator that uses the function. Next you create an indextype, which associates the implementation type with the operator. Finally, you create the domain index using this clause. Please refer to Appendix E, "Examples", which contains an example of creating a simple domain index, including all of these operations.

index_expr

In the *index_expr* (in *table_index_clause*), specify the table columns or object attributes on which the index is defined. You can define multiple domain indexes on a single column only if the underlying indextypes are different and the indextypes support a disjoint set of user-defined operators.

Restrictions on Domain Indexes

Domain indexes are subject to the following restrictions:

- The *index_expr* (in *table_index_clause*) can specify only a single column, and the column cannot be of datatype `REF`, varray, nested table, `LONG`, or `LONG RAW`.
- You cannot create a bitmap or unique domain index.
- You cannot create a domain index on a temporary table.

indextype

For *indextype*, specify the name of the indextype. This name should be a valid schema object that has already been created.

If you have installed Oracle Text, you can use various built-in indextypes to create Oracle Text domain indexes. For more information on Oracle Text and the indexes it uses, please refer to *Oracle Text Reference*.

See Also:

parallel_clause
Use the *parallel_clause* to parallelize creation of the domain index. For a nonpartitioned domain index, Oracle Database passes the explicit or default degree of parallelism to the ODCIIndexCreate cartridge routine, which in turn establishes parallelism for the index.

See Also:

*Oracle Data Cartridge Developer's Guide*for complete information on the Oracle Data Cartridge Interface (ODCI) routines

PARAMETERS
In the PARAMETERS clause, specify the parameter string that is passed uninterpreted to the appropriate ODCI indextype routine. The maximum length of the parameter string is 1000 characters.

When you specify this clause at the top level of the syntax, the parameters become the default parameters for the index partitions. If you specify this clause as part of the LOCAL [PARTITION] clause, you override any default parameters with parameters for the individual partition.

After the domain index is created, Oracle Database invokes the appropriate ODCI routine. If the routine does not return successfully, the domain index is marked FAILED. The only operations supported on an failed domain index are DROP INDEX and (for non-local indexes) REBUILD INDEX.

See Also:

*Oracle Data Cartridge Developer's Guide*for information on the Oracle Data Cartridge Interface (ODCI) routines

bitmap_join_index_clause

Use the *bitmap_join_index_clause* to define a **bitmap join index**. A bitmap join index is defined on a single table. For an index key made up of dimension table columns, it stores the fact table rowids corresponding to that key. In a data warehousing environment, the table on which the index is defined is commonly referred to as a **fact table**, and the tables with which this table is joined are commonly referred to as **dimension tables**. However, a star schema is not a requirement for creating a join index.

ON
In the ON clause, first specify the fact table, and then inside the parentheses specify the columns of the dimension tables on which the index is defined.

FROM

In the `FROM` clause, specify the joined tables.

WHERE
In the `WHERE` clause, specify the join condition.

If the underlying fact table is partitioned, you must also specify one of the *`local_partitioned_index`* clauses (see *local_partitioned_index*).

Restrictions on Bitmap Join Indexes
In addition to the restrictions on bitmap indexes in general (see BITMAP), the following restrictions apply to bitmap join indexes:

- You cannot create a bitmap join index on an index-organized table or a temporary table.
- No table may appear twice in the `FROM` clause.
- You cannot create a function-based join index.
- The dimension table columns must be either primary key columns or have unique constraints.
- If a dimension table has a composite primary key, each column in the primary key must be part of the join.
- You cannot specify the *`local_index_clauses`* unless the fact table is partitioned.

See Also:

*Oracle Data Warehousing Guide*for information on fact and dimension tables and on using bitmap indexes in a data warehousing environment

Examples

General Index Examples

Creating an Index: Example
The following statement shows how the sample index `ord_customer_ix` on the `customer_id` column of the sample table `oe.orders` was created:

```
CREATE INDEX ord_customer_ix ON orders (customer_id);
```

Compressing an Index: Example
To create the `ord_customer_ix_demo` index with the `COMPRESS` clause, you might issue the following statement:

```
CREATE INDEX ord_customer_ix_demo ON orders (customer_id, sales_rep_id) COMPRESS 1;
```

The index will compress repeated occurrences of `customer_id` column values.

Creating an Index in NOLOGGING Mode: Example
If the sample table `orders` had been created using a fast parallel load (so all rows were already sorted), you could issue the following statement to quickly create an index.

```
/* Unless you first sort the table oe.orders, this example fails because you cannot specify NOSORT unless the base table is
already sorted. */ CREATE INDEX ord_customer_ix_demo ON orders (order_mode) NOSORT NOLOGGING;
```

Creating a Cluster Index: Example
To create an index for the `personnel` cluster, which was created in "Creating a Cluster: Example", issue the following statement:

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

No index columns are specified, because cluster indexes are automatically built on all the columns of the cluster key. For cluster indexes, all rows are indexed.

Creating an Index on an XMLType Table: Example
The following example creates an index on the area element of the `xwarehouses` table (created in "XMLType Table Examples"):

```
CREATE INDEX area_index ON xwarehouses e (EXTRACTVALUE(VALUE(e),'/Warehouse/Area'));
```

Such an index would greatly improve the performance of queries that select from the table based on, for example, the square footage of a warehouse, as shown in this statement:

```
SELECT e.getClobVal() AS warehouse FROM xwarehouses e WHERE EXISTSNODE(VALUE(e),'/Warehouse[Area>50000]') = 1;
```

See Also:

EXISTSNODEand VALUE

Function-Based Index Examples

The following examples show how to create and use function-based indexes.

Creating a Function-Based Index: Example

The following statement creates a function-based index on the `employees` table based on an uppercase evaluation of the `last_name` column:

```
CREATE INDEX upper_ix ON employees (UPPER(last_name));
```

See the "Prerequisites"for the privileges and parameter settings required when creating function-based indexes.

To ensure that Oracle Database will use the index rather than performing a full table scan, be sure that the value returned by the function is not null in subsequent queries. For example, this statement is guaranteed to use the index:

```
SELECT first_name, last_name FROM employees WHERE UPPER(last_name) IS NOT NULL ORDER BY UPPER(last_name);
```

Without the `WHERE` clause, Oracle Database may perform a full table scan.

In the next statements showing index creation and subsequent query, Oracle Database will use index `income_ix` even though the columns are in reverse order in the query:

```
CREATE INDEX income_ix ON employees(salary + (salary*commission_pct)); SELECT first_name||' '||last_name "Name" FROM employees
WHERE (salary*commission_pct) + salary > 15000;
```

Creating a Function-Based Index on a LOB Column: Example

The following statement uses the function created in "Using a Packaged Procedure in a Function: Example"to create a function-based index on a LOB column in the sample `pm` schema. The example then collects statistics on the function-based index and selects rows from the sample table `print_media` where that `CLOB` column has fewer than 1000 characters.

```
CREATE INDEX src_idx ON print_media(text_length(ad_sourcetext)); ANALYZE INDEX src_idx COMPUTE STATISTICS; SELECT product_id FROM print_media WHERE text_length(ad_sourcetext) < 1000; PRODUCT_ID ---------- 3060 2056 3106 2268
```

Creating a Function-based Index on a Type Method: Example

This example entails an object type `rectangle` containing two number attributes: `length` and `width`. The `area()` method computes the area of the rectangle.

```
CREATE TYPE rectangle AS OBJECT ( length NUMBER, width NUMBER, MEMBER FUNCTION area RETURN NUMBER DETERMINISTIC ); CREATE OR REPLACE TYPE BODY rectangle AS MEMBER FUNCTION area RETURN NUMBER IS BEGIN RETURN (length*width); END; END;
```

Now, if you create a table `rect_tab` of type `rectangle`, you can create a function-based index on the `area()` method as follows:

```
CREATE TABLE rect_tab OF rectangle; CREATE INDEX area_idx ON rect_tab x (x.area());
```

You can use this index efficiently to evaluate a query of the form:

```
SELECT * FROM rect_tab x WHERE x.area() > 100;
```

Using a Function-based Index to Define Conditional Uniqueness: Example

The following statement creates a unique function-based index on the `oe.orders` table that prevents a customer from taking advantage of promotion ID 2 ("blowout sale") more than once:

```
CREATE UNIQUE INDEX promo_ix ON orders (CASE WHEN promotion_id =2 THEN customer_id ELSE NULL END, CASE WHEN promotion_id = 2 THEN promotion_id ELSE NULL END); INSERT INTO orders (order_id, order_date, customer_id, order_total, promotion_id) VALUES (2459, systimestamp, 106, 251, 2); 1 row created. INSERT INTO orders (order_id, order_date, customer_id, order_total, promotion_id) VALUES (2460, systimestamp+1, 106, 110, 2); insert into orders (order_id, order_date, customer_id, order_total, promotion_id) * ERROR at line 1: ORA-00001: unique constraint (OE.PROMO_IX) violated
```

The objective is to remove from the index any rows where the `promotion_id` is not equal to 2. Oracle Database does not store in the index any rows where all the keys are `NULL`. Therefore, in this example, we map both `customer_id` and `promotion_id` to `NULL` unless promotion_id is equal to 2. The result is that the index constraint is violated only if promotion_id is equal to 2 for two rows with the same `customer_id` value.

Partitioned Index Examples

Creating a Range-Partitioned Global Index: Example
The following statement creates a global prefixed index `cost_ix` on the sample table `sh.sales` with three partitions that divide the range of costs into three groups:

```
CREATE INDEX cost_ix ON sales (amount_sold) GLOBAL PARTITION BY RANGE (amount_sold) (PARTITION p1 VALUES LESS THAN (1000),
PARTITION p2 VALUES LESS THAN (2500), PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

Creating a Hash-Partitioned Global Index: Example
The following statement creates a hash-partitioned global index `cust_last_name_ix` on the sample table `sh.customers` with four partitions:

```
CREATE INDEX cust_last_name_ix ON customers (cust_last_name) GLOBAL PARTITION BY HASH (cust_last_name) PARTITIONS 4;
```

Creating an Index on a Hash-Partitioned Table: Example
The following statement creates a local index on the `product_id` column of the `hash_products` partitioned table (which was created in "Hash Partitioning Example"). The `STORE IN` clause immediately following `LOCAL` indicates that `hash_products` is hash partitioned. Oracle Database will distribute the hash partitions between the `tbs1` and `tbs2` tablespaces:

```
CREATE INDEX prod_idx ON hash_products(product_id) LOCAL STORE IN (tbs_01, tbs_02);
```

The creator of the index must have quota on the tablespaces specified. See CREATE TABLESPACEfor examples that create tablespaces `tbs_1` and `tbs_2`.

Creating an Index on a Composite-Partitioned Table: Example
The following statement creates a local index on the `composite_sales` table, which was created in "Composite-Partitioned Table Examples". The `STORAGE` clause specifies default storage attributes for the index. However, this default is overridden for the five subpartitions of partitions `q3_2000` and `q4_2000`, because separate `TABLESPACE` storage is specified.

The creator of the index must have quota on the tablespaces specified. See CREATE TABLESPACEfor examples that create tablespaces `tbs_1` and `tbs_2`.

```
CREATE INDEX sales_ix ON composite_sales(time_id, prod_id) STORAGE (INITIAL 1M MAXEXTENTS UNLIMITED) LOCAL (PARTITION q1_1998,
PARTITION q2_1998, PARTITION q3_1998, PARTITION q4_1998, PARTITION q1_1999, PARTITION q2_1999, PARTITION q3_1999, PARTITION
q4_1999, PARTITION q1_2000, PARTITION q2_2000 (SUBPARTITION pq2001, SUBPARTITION pq2002, SUBPARTITION pq2003, SUBPARTITION pq2004,
SUBPARTITION pq2005, SUBPARTITION pq2006, SUBPARTITION pq2007, SUBPARTITION pq2008), PARTITION q3_2000 (SUBPARTITION c1 TABLESPACE
tbs_02, SUBPARTITION c2 TABLESPACE tbs_02, SUBPARTITION c3 TABLESPACE tbs_02, SUBPARTITION c4 TABLESPACE tbs_02, SUBPARTITION c5
TABLESPACE tbs_02), PARTITION q4_2000 (SUBPARTITION pq4001 TABLESPACE tbs_03, SUBPARTITION pq4002 TABLESPACE tbs_03, SUBPARTITION
pq4003 TABLESPACE tbs_03, SUBPARTITION pq4004 TABLESPACE tbs_03) );
```

Bitmap Index Example

The following creates a bitmap join index on the table `oe.hash_products`, which was created in "Hash Partitioning Example":

```
CREATE BITMAP INDEX product_bm_ix ON hash_products(list_price) TABLESPACE tbs_1 LOCAL(PARTITION ix_p1 TABLESPACE tbs_02, PARTITION
ix_p2, PARTITION ix_p3 TABLESPACE tbs_03, PARTITION ix_p4, PARTITION ix_p5 TABLESPACE tbs_04 );
```

Because `hash_products` is a partitioned table, the bitmap join index must be locally partitioned. In this example, the user must have quota on tablespaces specified. See CREATE TABLESPACEfor examples that create tablespaces `tbs_2`, `tbs_3`, and `tbs_4`.

Indexes on Nested Tables: Example

The sample table `pm.print_media` contains a nested table column `ad_textdocs_ntab`, which is stored in storage table `textdocs_nestedtab`. The following example creates a unique index on storage table `textdocs_nestedtab`:

```
CREATE UNIQUE INDEX nested_tab_ix ON textdocs_nestedtab(NESTED_TABLE_ID, document_typ);
```

Including pseudocolumn `NESTED_TABLE_ID` ensures distinct rows in nested table column `ad_textdocs_ntab`.

Indexing on Substitutable Columns: Examples

You can build an index on attributes of the declared type of a substitutable column. In addition, you can reference the subtype attributes by using the appropriate `TREAT` function. The following example uses the table `books`, which is created in "Substitutable Table and Column Examples". The statement creates an index on the `salary` attribute of all employee authors in the `books` table:

```
CREATE INDEX salary_i ON books (TREAT(author AS employee_t).salary);
```

The target type in the argument of the `TREAT` function must be the type that added the attribute being referenced. In the example, the target of `TREAT` is `employee_t`, which is the type that added the `salary` attribute.

If this condition is not satisfied, then Oracle Database interprets the `TREAT` function as any functional expression and creates the index as a function-based index. For example, the following statement creates a function-based index on the `salary` attribute of part-time employees, assigning nulls to instances of all other types in the type hierarchy.

```
CREATE INDEX salary_func_i ON persons p (TREAT(VALUE(p) AS part_time_emp_t).salary);
```

You can also build an index on the type-discriminant column underlying a substitutable column by using the `SYS_TYPEID` function.

> **Note:**
>
> Oracle Database uses the type-discriminant column to evaluate queries that involve the `IS OF` *`type`* condition. The cardinality of the typeid column is normally low, so Oracle recommends that you build a bitmap index in this situation.

The following statement creates a bitmap index on the typeid of the author column of the books table:

```
CREATE BITMAP INDEX typeid_i ON books (SYS_TYPEID(author));
```

See Also:

- "Type Hierarchy Example"to see the creation of the type hierarchy underlying the `books` table
- the functions TREATand SYS_TYPEIDand the condition "IS OF *type* Condition"